Algorithms and Data Structures 2
CS 1501

Spring 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming deadlines:
  - Assignment 3 and 4 late deadline on 4/20
  - Lab 12 due on 4/22
  - Assignment 5 due on 5/2 (**no late deadline**)
  - Bonus Opportunities:
    - Bonus Lab due on 5/2
    - Bonus Homework due on 5/2
    - 1 bonus point for entire class when response rate >= 80%
      - Currently at ~16%
      - Deadline is Sunday 4/24
- CourseMIRROR Post-survey
  - https://purdue.ca1.qualtrics.com/jfe/form/SV_8zRF6IAtHSaRKJg

Previous lecture …

- Master Method
- Unbounded Knapsack Problem
  - Dynamic Programming algorithm
  - Greedy algorithm

CourseMIRROR Reflections (most confusing)

- how to determine the values in max value array
- Brief review of calculating runtime through master theorem
- I would like some more explanation of the recurrence relation cases
- Why T(n) for Binary Search is log2(n) instead of lg(n), as both non-recursive and recursive parts have constant time complexity.
- when we cannot use the master theorem (when it doesn’t meet condition #3 from the slides about being polynomially larger/smaller)

CourseMIRROR Reflections (most interesting)

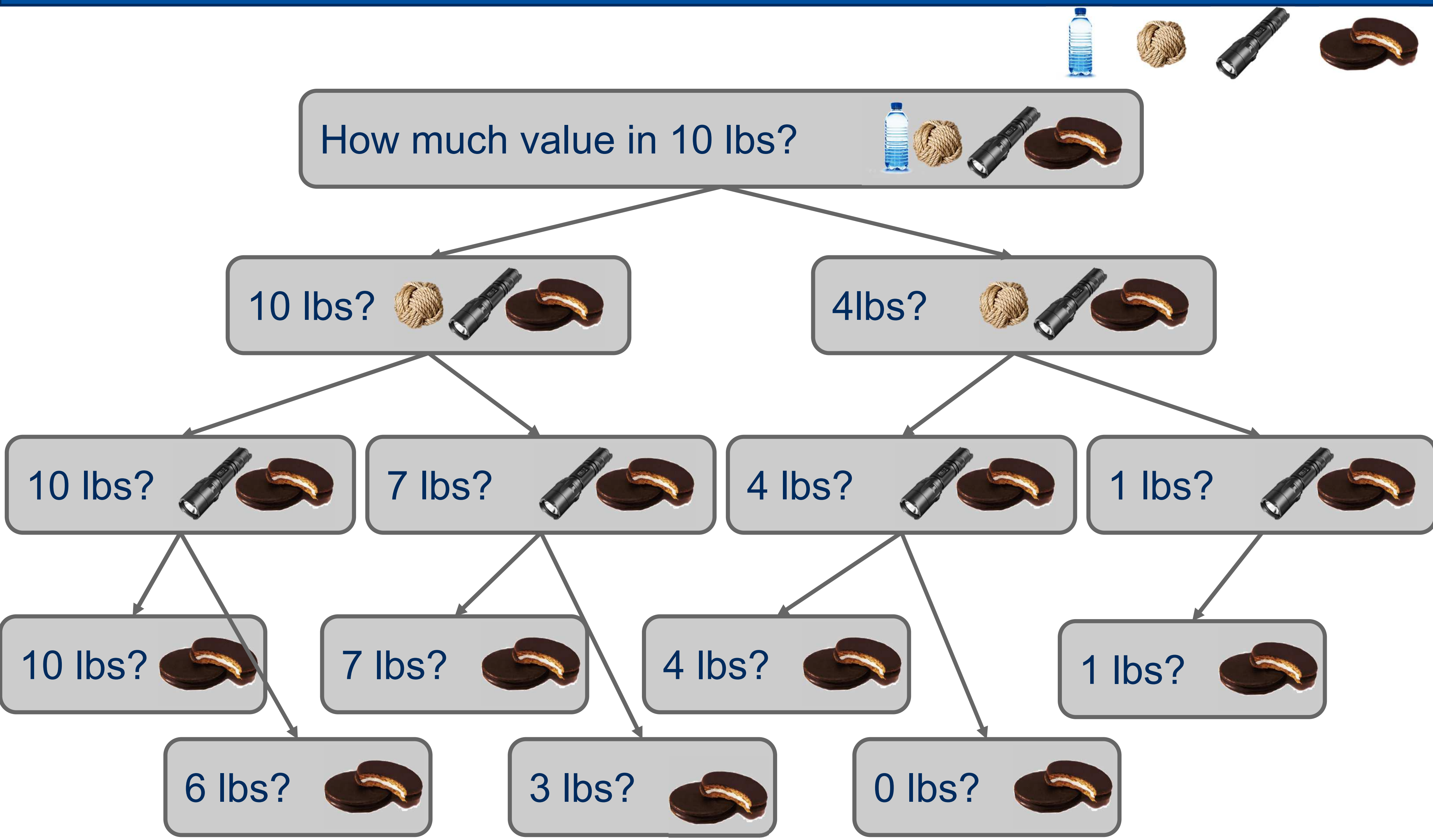
- How the greedy algorithm led to an incorrect answer for the knapsack problem
- I found the use of the master method to be much clearer and very useful
- Examples for Master Theorem
- I enjoyed learning about the knapsack problem and dynamic programming
- bottom up dynamic programming technique

Dynamic Programming Example 1: The 0/1 knapsack problem

- What if we have a finite set of items that each has a weight and value?
  - Two choices for each item:
    - Goes in the knapsack
    - Is left out

0/1 Recursive solution

weight:	6	3	4	2
value:	30	14	16	9



Recursive solution

```
int knapSack(int[] wt, int[] val, int L, int n) {
    if (n == 0 || L == 0) { return 0 };
    if (wt[n-1] > L) {
        return knapSack(wt, val, L, n-1)
    }
    else {
        return max( val[n-1] + knapSack(wt, val, L-wt[n-1], n-1),
                                knapSack(wt, val, L, n-1)
                              );
    }
}
```

The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
K[n+1][L+1]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0											
1											
2											
3											
4											

K[i][l] is the best (max) value when only the first ***i*** items are available and only ***l*** lbs remain in the knapsack

The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
1	**0**										
2	**0**										
3	**0**										
4	**0**										

K[i][l] is the best (max) value when only the first ***i*** items are available and only ***l*** lbs remain in the knapsack

The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0					
2	0										
3	0										
4	0										

The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0										
3	0										
4	0										

The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0								
3	0										
4	0										

The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0	14	14	14	30	30	30	44	44
3	0										
4	0										

The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0	14	14	14	30	30	30	44	44
3	0	0	0	0	16						
4	0										

The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0	14	14	14	30	30	30	44	44
3	0	0	0	0	16	16	30	30	30	44	46
4	0										

The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0	14	14	14	30	30	30	44	44
3	0	0	0	0	16	16	30	30	30	44	46
4	0	0									

The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0	14	14	14	30	30	30	44	44
3	0	0	0	0	16	16	30	30	30	44	46
4	0	0	9	9	16	16	30	30	39	44	46

The 0/1 knapsack dynamic programming solution

```
int knapSack(int wt[], int val[], int L, int n) {
    int[][] K = new int[n+1][L+1];
    for (int i = 0; i <= n; i++) {
        for (int l = 0; l <= L; l++) {
            if (i==0 || l==0){ K[i][l] = 0 };
            else if (wt[i-1] > l){ K[i][l] = K[i-1][l] };
            else {
                K[i][l] = max(val[i-1] + K[i-1][l-wt[i-1]],
                                          K[i-1][l]);
            }
        }
    }
    return K[n][L];
}
```

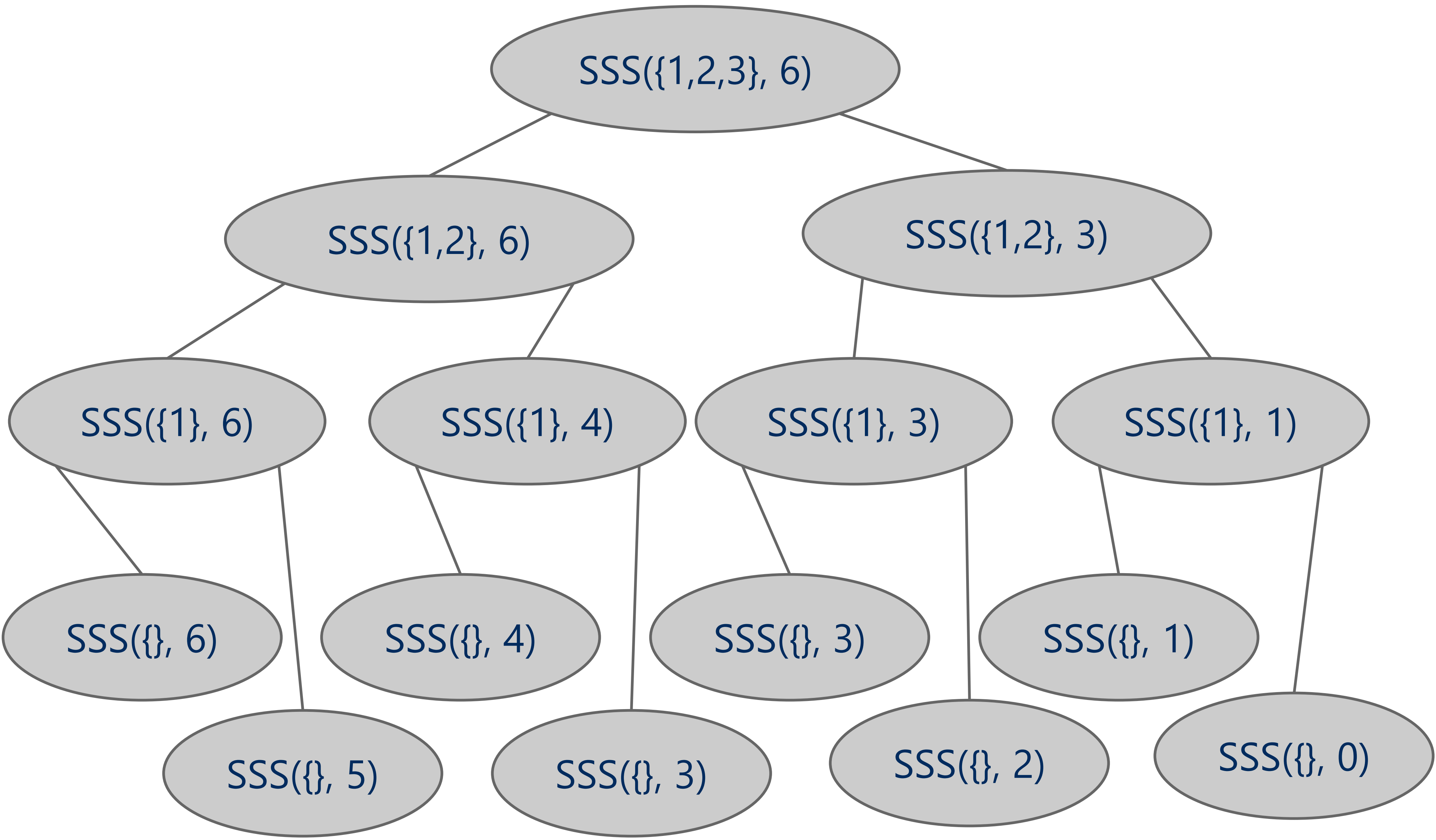
To review...

- Questions to ask in finding dynamic programming solutions:
  - Does the problem have optimal substructure?
    - Can solve the problem by splitting it into smaller problems?
    - Can you identify subproblems that build up to a solution?
  - Does the problem have overlapping subproblems?
    - Where would you find yourself recomputing values?
      - How can you save and reuse these values?

Dynamic Programming Example 2: Subset sum

- Given a set of non-negative integers S and a value k, is there a subset of S that sums to exactly k?

Subset sum calls



Subset sum recursive solution

```
boolean SSS(int set[], int sum, int n) {
    if (sum == 0)
          return true;
    if (sum != 0 && n == 0)
          return false;
    if (set[n-1] > sum)
          return SSS(set, sum, n-1);
    return SSS(set, sum, n-1)
          || SSS(set, sum-set[n-1], n-1);
}
```

- What would a dynamic programming table look like?

Subset sum bottom-up dynamic programming

```
boolean SSS(int set[], int sum, int n) {
    boolean[][] subset = new boolean[sum+1][n+1];
    for (int i = 0; i <= n; i++) subset[0][i] = true;
    for (int i = 1; i <= sum; i++) subset[i][0] = false;
    for (int i = 1; i <= sum; i++) {
       for (int j = 1; j <= n; j++) {
              subset[i][j] = subset[i][j-1];
              if (i >= set[j-1])
                     subset[i][j] ||= subset[i - set[j-1]][j-1];
       }
    }
    return subset[sum][n];
}
```

Example 3: Change making problem

Consider a currency with n different denominations of coins d_1, d_2, …, d_n. What is the minimum number of coins needed to make up a given value k?

Solution Attempt

If you were working as a cashier, what would your algorithm be to solve this problem?

… But wait …

- Does our greedy change making algorithm solve the change making problem?
  - For US currency…
  - But what about a currency composed of pennies (1 cent), thrickels (3 cents), and fourters (4 cents)?
    - What denominations would it pick for k=6?

So, how can we solve the change making problem optimally?

We will see a dynamic programming algorithm in the recitation of this week.

Problem of the Day: Travelling Salesman problem

Given a list of cities and the distances between **each pair** of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?

But first, something completely different...

- Some computational problems are *unsolvable*
  - No algorithm can be written that will always produce the correct output
  - One example is the *halting problem*
    - Given a program and an input, will the program halt?
      - Can we write an algorithm to determine whether any program/input pair will halt?

Intractable problems

- Solvable, but require too much time to solve to be practically solved for modest input sizes
    - Listing all of the subsets of a set:
        - $\Theta(2^n)$
    - Listing all of the permutations of a sequence:
        - $\Theta(n!)$

Polynomial time algorithms

- Most of the algorithms we've covered so far this term
  - Also the most practically useful of the three classes we've just covered...
- Largest term in the runtime is a simple power with a constant exponent
  - E.g., n^2
  - Or a power times a logarithm
    - E.g., $n \lg n$

Consider the following

- The shortest path problem
  - Easily solved in polynomial time
- The longest path problem
  - How long would it take us to find the longest path between two points in a graph?

What if a problem doesn't fall into one of our three categories?

- It can be solved
- There is no proof that a solution requires exponential time
  - … yet
- There is no valid solution that runs in polynomial time
  - … yet

P vs NP

- P
  - The set of problems that can be solved by deterministic algorithms in polynomial time
- NP
  - The set of problems that can be solved by non-deterministic algorithms in polynomial time
    - i.e., solution from a non-deterministic algorithm can be verified in polynomial time

Deterministic vs non-deterministic algorithms

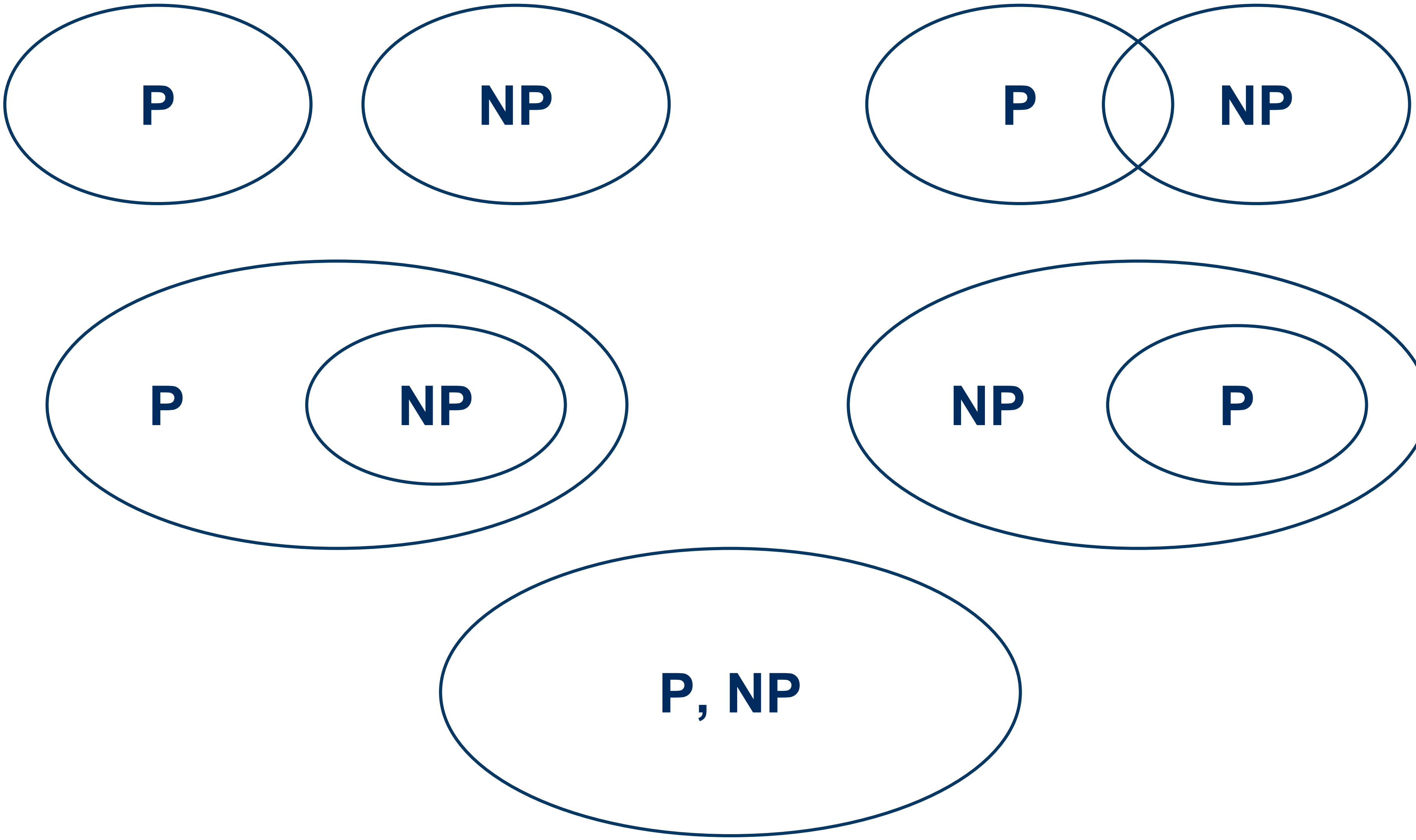
- Deterministic
  - At any point during the run of the program, given the current instruction and input, we can predict the next instruction
  - Running the same program on the same input produces the same sequence of executed instructions
- Non-deterministic
  - A **conceptual** algorithm with more than one allowed step at certain times and which always takes the right or best step
    - Conceptually, could run on a deterministic computer with unlimited parallel processors
      - Would be as fast as always choosing the right step

Non-deterministic algorithms

- Array search:
  - Linear search:
    - $\Theta(n)$
  - Binary search:
    - $\Theta(\lg n)$
  - Non-deterministic search algorithm:
    - $\Theta(1)$

So we can group problems into P and NP...

- 5 options for how the sets P and NP intersect:



Are any of these clearly impossible?

- Why?

Please submit your reflections by using the CourseMIRROR App

If you are having a problem with CourseMIRROR, please send an email to coursemirror.development@gmail.com